Converting MiniZinc to FlatZinc
Version 1.1

Nicholas Nethercote

1 Introduction

This document specifies how to convert MiniZinc to FlatZinc.

We will use the MiniZinc model and example data for a restricted job shop scheduling problem in Figures 1 and 2 as a running example.

This MiniZinc model instance is translated into the FlatZinc code shown in Figure 3. Line 30 is the original two-dimensional array of decision variables, mapped to a zero-indexed one-dimensional array. Lines 32–35 are variables introduced by Boolean decomposition. Lines 36–45 are the constraints. Lines 37 and 39 result from line 12, lines 38 and 40 result from line 13, and lines 41–46 result from lines 14–15 and 7–8.

```
----------------
ANNOTATIONS

Handling annotations is crucial, and not easy.  For every translation step
we have to specify how annotations are treated.

Ideally, for every translation step there will be a single way to handle
annotations which is suitable in all cases.  However, this may not provide
enough control -- it is likely that some annotations will need different
treatment to others.

One way of providing more control is to divide annotations into two or more
classes.  Each class of annotation will be treated in a particular way in
each translation step.  For example:

  prop_on_introduced_vars         annotation a1;
  prop_on_introduced_constraints  annotation a2;
  prop_on_introduced_vars_cons    annotation a3;
  no_prop                         annotation a4; (default behaviour)

A related alternative would be to classify them according to the elements
they can annotate, eg:

  annotation [var_decl] f1(...)
  annotation [expr] f2(...)
  annotation [var_decl, expr] f3(...)
  annotation [solve] f4(...)

This approach has the additional advantage of allowing some extra
compile-time sanity checking of annotations.
```

```
0    % (square) job shop scheduling in MiniZinc
1    int: size;                                  % size of problem
2    array [1..size,1..size] of int: d;          % task durations
3    int: total = sum(i,j in 1..size) (d[i,j]);  % total duration
4    array [1..size,1..size] of var 0..total: s; % start times
5    var 0..total: end;                          % total end time
6
7    predicate no_overlap(var int:s1, int:d1, var int:s2, int:d2) =
8        s1 + d1 <= s2 \/ s2 + d2 <= s1;
9
10   constraint
11       forall(i in 1..size) (
12           forall(j in 1..size-1) (s[i,j] + d[i,j] <= s[i,j+1]) /\
13           s[i,size] + d[i,size] <= end /\
14           forall(j,k in 1..size where j < k) (
15               no_overlap(s[j,i], d[j,i], s[k,i], d[k,i])
16           )
17       );
18
19   solve minimize end;
```

Figure 1: MiniZinc model (`jobshop.mzn`) for the job shop problem.

```
20   size = 2;
21   d = [| 2,5
22        | 3,4 ];
```

Figure 2: MiniZinc data (`jobshop2x2.data`) for the job shop problem.

```
Or, if that is still too crude, the most control would be provided by
identifying a number of possible annotation approaches for each translation
step, and for each annotation, marking it with the approach used for every
step.  For example:

  annotation a1 [parameter_substitution: parameter_substitution_approach_1,
                 builtins_evaluation: builtins_evaluation_approach_2,
                 ...]

Hopefully this won't be needed, as it would be cumbersome.  If we're lucky,
a single approach will suffice, but we won't know until the described
approach is implemented and we can experiment with it.

----------------
ANNOTATION BASICS

Annotations can appear on variable declarations (both global and let-local),
expressions, and solve items.  We call the thing annotated the "annotatee".
Annotations on solve items are easy, as solve items are left untouched by
the translation.  Annotations on variable declarations and (in particular)
expressions are the interesting ones.

When a translation step rewrites some MiniZinc code, there are three
```

```
30   array[0..3] of var 0..14: s;
31   var 0..14: end;
32   var bool: b1;
33   var bool: b2;
34   var bool: b3;
35   var bool: b4;
36   constraint  int_lin_le     ([1,-1], [s[0], s[1]], -2);
37   constraint  int_lin_le     ([1,-1], [s[1], end ], -5);
38   constraint  int_lin_le     ([1,-1], [s[2], s[3]], -3);
39   constraint  int_lin_le     ([1,-1], [s[3], end ], -4);
40   constraint  int_lin_le_reif([1,-1], [s[0], s[2]], -2, b1);
41   constraint  int_lin_le_reif([1,-1], [s[2], s[0]], -3, b2);
42   constraint  bool_or(b1, b2, true);
43   constraint  int_lin_le_reif([1,-1], [s[1], s[3]], -5, b3);
44   constraint  int_lin_le_reif([1,-1], [s[3], s[1]], -4, b4);
45   constraint  bool_or(b3, b4, true);
46   solve minimize end;
```

Figure 3: FlatZinc translation of the MiniZinc job shop model.

```
possible treatments for each annotations.
- The annotation can be copied zero times (ie. deleted).
- The annotation can be copied once.
- The annotation can be copied multiple times.

----------------
ANNOTATION HANDLING PRINCIPLES

It's useful to identify some overall principles to annotation handling, to
guide the decisions about how they are handled.  They may not be perfect,
but at least they provide a foundation for consistency.  The handling of
every possible annotation in every translation step is justified by at least
one of them.

0. delete-if-deleted
--------------------
If an annotatee doesn't contribute in any way to the result of the
translation, any annotation on it should not be copied to the result.  This
just obviously seems like the right thing to do.

1. delete-if-fixed
------------------
If a fixed annotatee is copied, its annotation is deleted.

Annotations are mostly used to guide solving.  Therefore, they are most
relevant to decision variables and constraints, and annotations on fixed
variables (parameters) and fixed expressions don't seem very useful.  For
example:

  int: x::A1 = 3::A2;

It is unclear what interesting properties A1 and A2 could specify here.
```

Furthermore, many fixed variables and expressions are combined, greatly simplified or removed during flattening.

Therefore, it seems reasonable that annotations on fixed variables and expressions are deleted during translation, and any fixed variables and expression introduced by a translation step are left unannotated. This has the additional benefit of reducing the number of annotation cases that must be handled.

2. maybe-copy-if-copied

If a non-fixed annotatee is copied (unchanged), its annotation is copied (unchanged) with it. We use the term "maybe-copying" for copying an annotation to the RHS if its annotatee is non-fixed, and deleting it otherwise.

This is simple and just seems obvious as the right thing to do.

3. maybe-copy-if-copied-with-subs

If a non-fixed annotatee is copied (with substitutions), its annotation is copied (with substitutions) with it.

This is an obvious extension of maybe-copy-if-copied.

4. maybe-copy-if-modified

If a non-fixed annotatee is modified, its annotation is copied to the "corresponding position" in the result.

This is like maybe-copy-if-copied, but open to more interpretation, because "corresponding position" isn't always an exact concept.

5. maybe-copy-multiple

If a non-fixed annotatee becomes multiple annotatees, its annotation is copied multiple times.

This also seems reasonable, albeit open to even more interpretation than maybe-copy-if-modified.

6. maybe-copy-to-new-variables

When a variable is introduced by a translation step, any annotation on the responsible expression is copied to the variable declaration. This seems to be a useful behaviour.

XXX: but it might be one that requires some control over it; it assumes that annotations make sense on both variables declarations and expressions. If we classified annotation by the things they can appear on, useful (var-decls, exprs, var-decls-and-exprs, solve items); a 'exprs' annotation wouldn't be copied to the declaration, but a var-decls-and-exprs one would be).

```
7. mark-new-variables
---------------------
New variables are annotated with 'var_is_introduced'.  This means they can
be identified by the FlatZinc implementation, which can be helpful, e.g. for
deciding which variable values to print once a solution is found.

8. ad hoc
---------
In a small number of cases, the chosen approach doesn't follow any of the
above principles, and a case-specific justification is used.

----------------
```

2 The Translation

The translation has two parts: flattening, and the rest.

```
----------------
For each translation step, we use Cadmium-like syntax to specify its
behaviour, i.e. with a left-hand side (LHS) and right-hand side (RHS)
showing the code before and after the translation step.  For example:

    LHS  -->  RHS

means that LHS is translated to RHS.

We use the following syntactical conventions.
- E, E1, E2, ... are arbitrary expressions.
- A, A1, A2, B, B1, B2, C, C1, C2, D, D1, D2,... are annotations.
- T, T1, T2, ... are type-inst expressions.
- X, X1, ... are variables or predicate/function arguments.
- (...X...) is an expression containing X.
- F, F1, F2, G, P, ... are operations (functions/predicates).
- 'P' in front of an annotatee means it is definitely fixed (eg. PE is a
  fixed expression, PT is a fixed type-inst expression).  'P' in front of an
  annotation means it is attached to a definitely fixed annotatee.

Note that all annotations shown on the RHS of a rule are only copied if
their annotatee is non-fixed.

When a rule includes a conjunctive context, the RHS shows how the context is
also modified.  For example:

    A \ B  -->  C \ D

means "when B is in the conjunctive context of A, A translates into C and
B translates into D".
----------------
```

2.1 Flattening

Flattening involves the following simple steps that statically evaluate (or *reduce*) the model and data as much as possible. There is no fixed order to the steps because

some enable others, which can then enable further application of previously applied steps. Therefore, they must be repeatedly applied, e.g. by iterating until a fixpoint is reached, or by re-flattening child nodes of expressions that have been flattened.

2.1.1 Stand-alone assignment removal.

This step removes each stand-alone assignment by merging it with the appropriate variable declaration.

```
----------------
T: X::A1 \ X = E::A2;  -->  T: X::A1 = E::A2 \ <empty>

Annotations rationale.
- A1, A2: maybe-copy-if-copied.

XXX: here and everywhere else:  if any annotations have constrained fixed
arguments, the argument values should be checked to make sure they satisfy
the constraints
----------------
```

2.1.2 Parameter substitution.

For every declaration of a global or let-local scalar parameter that is assigned a constant literal value, this step: (a) checks that the value satisfies any type-inst constraints on the parameter, and aborts if not; (b) substitutes the value for all uses of the parameter throughout the model; and (c) removes the declaration (in the let-local parameter case, if there are no other local variables remaining in the let expression, the let expression is replaced with its body).

```
----------------
% This assumes that PE satisfies any constraints on PT.
PT: X::PA1 = PE::PA2; \ X::PA3  -->  <empty> \ PE

Annotations rationale.
- PA1, PA2, PA3: delete-if-fixed.
----------------
```

For example, with `int: size = 2;` we substitute 2 for `size`, but `int: size = 2 * y;` would not be substituted until the right-hand side is fully reduced.

```
XXX: does this apply to arrays and sets, or just scalars?  Unclear.
If it does not, then some of the subsequent steps must work on both
constants literals *and* identifiers that are assigned constant literals
(and identifiers that are assigned identifiers that are assigned constant
literals...)  Eg. built-ins evaluation, comprehension unrolling.  Might be
easier to remove this step, and just make all the other steps work that way.

XXX: Should we also substitute decision variables, if they are assigned to?
Doing so would avoid the creation of some equality constraints.  What about
arrays -- it can cause code bloat, but it might be important in some cases.
It affects whether assignment is the same as equality...
If we do, some/all of the annotations will be preserved (maybe-copy-if-copied).
It's also trickier determining if the type-inst constraints are met, because
the RHS won't necessarily be fully flattened.  But the type-inst constraints
must be checked at some point, and no other translation step appears to do
so...
```

2.1.3 Built-ins evaluation.

This step evaluates all built-ins that have constant literal arguments.

```
----------------
F(PE1::PA1, PE2::PA2, ...):PB  -->  PE3

Annotations rationale.
- PA1, PA2, ..., PB: delete-if-fixed.
----------------
```

For example, `2-1` (from `size-1`, after parameter substitution) in our example becomes `1`.

2.1.4 Comprehension unrolling.

This step unrolls all set and array comprehensions, once the generator ranges are fully reduced to constant literals, replacing the generator variables in the expressions with literal values.

```
----------------
% Set comprehensions.
{(...PE::PA1(PE)...)::PA2(PE)
| PE in [PE1::PB1, PE2::PB2, ...] where (...PE::PC1...)::PC2}::PD
    -->  {(...PE1...), (...PE2...), ...}

Annotations rationale.
- All annotations: delete-if-fixed -- sets cannot contain non-fixed elements.


% Array comprehensions.
[(...PE::PA1(PE)...)::A2(PE)
| PE in [PE1::PB1, PE2::PB2, ...] where (...PE::PC1...)::PC2]::D
    -->  [(...PE1...)::A2(PE1), (...PE2...)::A2(PE2), ...]::D

Annotations rationale.
- PA1(PE), PB1, PB2, ..., PC1, PC2: delete-if-fixed.
- A2(PE): maybe-copy-multiple + maybe-copy-if-copied-with-subs.
- D: maybe-copy-if-modified.

Note.  When multiple generators are present, the translation step extends in
the obvious manner.
----------------
```

2.1.5 Compound built-in unrolling.

This step unrolls calls to compound built-ins (such as `sum` and `forall`; the full list is in Section 3.1) that have a (possibly non-constant) literal array argument by replacing them with multiple lower-level operations.

```
----------------
% Assumption: F is a compound built-in, G is its lower-level equivalent.
F([E1::A1, E2::A2, ...]::B)::C  -->  G(E1::A1::B, G(E2::A2::B, ...)::C)::C

Annotations rationale.
- A1, A2, ...: maybe-copy-if-copied.
```

```
- B: maybe-copy-multiple.  Note: when using the generator call syntactic
  sugar, this position cannot be annotated, so it's less useful than it
  appears.
- C: maybe-copy-multiple.  This one is important, as it ensures all the
  lower-level operations get the same annotation as the compound annotation.
----------------
```

We will use lines 11, 14 and 15 of Figure 1 as the starting point of a running example. They unroll to give the following conjunction (the first conjunct has `i=1`, `j=1` and `k=2`; the second has `i=2`, `j=1` and `k=2`).

```
no_overlap(s[1,1], d[1,1], s[2,1], d[2,1]) /\
no_overlap(s[1,2], d[1,2], s[2,2], d[2,2])
```

2.1.6 Fixed array access replacement.

This step replaces all array accesses involving constant literal indices and constant literal elements with the appropriate value. Furthermore, if all the accesses of an array variable have been replaced, its declaration and assignment can be removed.

```
----------------
PE has value [PE1::PA1, PE2::PA2, ...]::PB \ (PE::PC)[PE3::PA3]::PD
    -->  PE1 or PE2 or ...

Annotations rationale.
- All annotations: delete-if-fixed.
----------------
```

For example, our running example becomes:

```
no_overlap(s[1,1], 2, s[2,1], 3) /\
no_overlap(s[1,2], 5, s[2,2], 4)
```

```
XXX: What about if the access has fixed indices but a non-fixed element?  Do
them too?  (Then some annotations would be preserved.)  Some elements could
be '_', which would require anonymous variable naming to occur first?
```

2.1.7 If-then-else evaluation.

This step evaluates each if-then-else expression once its condition is fully reduced to a constant literal. This is always possible because if-then-else conditions must be fixed.

```
----------------
(if PE1::PA1 then E2::A2 else E3::A3 endif)::D  -->  E2::A2::D or E3::A3::D

Annotations rationale.
- PA1: delete-if-fixed.
- A2: maybe-copy-if-copied, A3: delete-if-deleted (or vice versa).
- D: maybe-copy-if-modified.
----------------
```

2.1.8 Predicate inlining.

This step replaces each call to a defined predicate with its body, substituting actual arguments for formal arguments. It also needs to check that fixed arguments satisfy their constraints, and abort if not. (Non-fixed arguments need not be checked, as they are checked at compile-time.)

Inlining is easy because predicates cannot be recursive, either directly or mutually. For predicates that call other predicates, the inlining can be done outside-in or inside-out. For example, in `p(q(x))` we can inline `p` first, then `q`, or vice versa. Calls to predicates lacking a definition (such as those in the MiniZinc globals library) are left as-is. Once all calls to a predicate have been inlined, it can be removed.

```
----------------
% The predicate is only removed once all calls have been inlined.
predicate P(T: X) = (...X::A1(X)...)::A2(X) \ P(E::B1)::B2
    -->  <empty> \ (...E::A1(X)::B1...)::A2(E)::B2

Note.  For predicates/functions with multiple arguments, this translation
step extends in the obvious manner.

Annotations rationale.
- A1(X): ad hoc.  Although X isn't copied to the result, maybe-copying (with
  substitutions) it seems the only sensible thing to do -- the
  only other obvious option is to discard it, which doesn't seem sensible.
- A2(X): maybe-copy-if-copied-with-subs.
- B1: maybe-copy-if-copied.
- B2: maybe-copy-if-modified.
----------------
```

For example, the first conjunct from our running example (after fixed array accesses to `d` have been flattened) becomes:

```
  s[1,1] + 2 <= s[2,1] \/ s[2,1] + 3 <= s[1,1]
```

2.1.9 Fixed objective removal

This step converts a `solve minimize` or `solve maximize` item to a `solve satisfy` item if the expression being minimized/maximized is fixed.

```
----------------
solve minimize PE::PA    --> solve satisfy;

Annotations rationale.
- All annotations: delete-if-fixed.
----------------
```

2.2 Post-flattening

After flattening, we apply the following steps once each, in the given order. Those steps marked with a '*' can be performed in more than one way, so their output depends on the exact details of the implementation, although the results should not vary greatly.

2.2.1 Let floating.

This step moves let-local decision variables to the top-level and renames them appropriately. The type-inst is unchanged.

```
----------------
(let { T: X::A1 = E::A2 } in (...X::B1...)::B2)::B3
    -->  T: X'::A1::B3 = E::A2; \ (...X'::B1...)::B2::B3

Annotations rationale.
- A1, B1, B2: maybe-copy-if-copied (modulo the renaming of X).
- A2: maybe-copy-if-copied.
- B3: maybe-copy-if-copied (the expression) + ad hoc (the variable
  declaration).  The latter is not obvious, but important -- it means that
  e.g. if a predicate call is marked with an expression, when that predicate
  is inlined, if it contains local variables they will end up with the
  annotation.
----------------
```

2.2.2 Boolean decomposition*.

This step decomposes all Boolean expressions that are not top-level conjunctions. It replaces each non-flat sub-expression with a new Boolean variable (also adding a declaration for each variable) that is initialised with the sub-expressions it replaced. This facilitates the later introduction of reified constraints.

```
----------------
% Assumption: E1 and E2 are non-flat expressions.
F(E1::A1, E2::A2)::B
    -->  var bool: X1::B::var_is_introduced;
         var bool: X2::B::var_is_introduced;
         constraint (E1::A1 = X1) /\ (E2::A2 = X2); \
         F(X1::A1, X2::A2)::B

Annotations rationale.
- A1, A2: maybe-copy-if-copied (the E1/E2 expressions) +
  maybe-copy-if-modified (the X1/X2 identifiers).
- B: maybe-copied (the expression) + maybe-copy-to-new-variables (the
  declarations).
- var_is_introduced: mark-new-variables.

XXX What about annotating the reified constraint?
This seems not to follow our assumption, that we should be able to annotate
every resulting item. Maybe the A1 should also go on the reified constraint?
This issue also appears with the other decomposition transformations.
----------------
```

For example, this expression:

```
s[1,1] + 2 <= s[2,1] \/ s[2,1] + 3 <= s[1,1]
```

becomes:

```
var bool: b1::var_is_introduced;
var bool: b2::var_is_introduced;
constraint ((s[1,1] + 2 <= s[2,1]) = b1) /\
```

```
            ((s[2,1] + 3 <= s[1,1]) = b2);
  ...
  ((b1 \/ b2) <-> true) /\
```

2.2.3 Numeric decomposition*.

This step decomposes numeric equations or inequations in a manner similar to Boolean decomposition, by renaming each non-linear, non-flat sub-expression with a new variable.

```
----------------
% Assumption:  E1 and E2 are non-flat expressions, and F is a non-linear
% operation.
F(E1::A1, E2::A2)::B
    -->  var int: X1::B::var_is_introduced;             % or var float
         var int: X2::B::var_is_introduced;             % or var float
         constraint (E1::A1 = X1) /\ (E2::A2 = X2); \
         F(X1::A1, X2::A2)::B

Annotations rationale.
- As for Boolean decomposition.
----------------
```

2.2.4 Set decomposition*.

This step decomposes compound set expressions into primitive set constraints in a manner similar to Boolean/numeric decomposition.

```
----------------
% Assumption: E1 and E2 are non-flat expressions.
% X1min..X1max is the smallest range that fits the element ranges of all the
% set sub-expressions of E1;  if one of the set sub-expressions is '_' a
% translation-time abort occurs, because no bounds can be computed.
% X2min..X2max is determined likewise from E2.
F(E1::A1, E2::A2)::B
    -->  var set of X1min..X1max: X1::B::var_is_introduced;
         var set of X2min..X2max: X2::B::var_is_introduced;
         constraint (E1::A1 = X1) /\ (E2::A2 = X2); \
         F(X1::A1, X2::A2)::B

Annotations rationale.
- As for Boolean decomposition.
----------------
```

2.2.5 (In)equality normalisation*.

This step normalises linear equations, inequations and disequations. It (a) moves sub-expressions so the right-hand side is constant; and (b) replaces negations with multiplications by -1. This facilitates the later introduction of linear constraints.

Sub-step (c) also applies to solve items that involve a linear expression.

```
----------------
% Where E1b and E2b are E1 and E2 with any constants removed and turned into
% k.
F(E1::A1, E2::A2)::B  -->  F(E1b::A1 + (-1)*E2b::A2, k)::B
```

```
Annotations rationale.
- A1, A2, B: maybe-copy-if-modified.
- No annotations on k and (-1): delete-if-fixed.
----------------
```

For example, the second conjunct from our running example becomes:

```
(s[1,1] + (-1)*s[2,1] <= -2) <-> b1
```

2.2.6 Array simplification.

This step simplifies all multi-dimensional array variables, 2d array literals, and calls to `array*d` to one-dimensional, zero-indexed equivalents, and also updates any remaining multi-dimensional array accesses accordingly.

```
----------------
% Multi-dimensional array declarations.
array[PE1::PA1, PE2::PA2, ...] of T: X::B  -->  array[PE3] of T: X::B;

Annotations rationale.
- PA1, PA2, ...: delete-if-fixed.
- B: maybe-copy-if-copied.


% 2d array literal.
[| E1::A1, E2::A2, ... |]  -->  [E1::A1, E2::A2, ...]

Annotations rationale.
- A1, A2, ...: maybe-copy-if-copied.


% Call to \texttt{array*d}.
array*d(PE1::PA1, PE2::PA2, ..., [E1::B1, E2::B2, ...])
    -->  [E1::B1, E2::B2...]

Annotations rationale.
- PA1, PA2, ...: delete-if-fixed.
- B1, B2, ...: maybe-copy-if-copied.


% Multi-dimensional array access.
(E::A)[E1::B1, E2::B2, ...]::C  -->  (E::A)[f(E1::B1, E2::B2, ...)]::C

Annotations rationale.
- A, B1, B2, ...: maybe-copy-if-copied.
- C: maybe-copy-if-modified.
----------------
```

For example, our running example becomes:

```
(s[0] + (-1)*s[2] <= -2) <-> b1
```

2.2.7 Anonymous variable naming.

This step replaces each anonymous variable ('_') with a newly introduced variable that has the same type, and also has any type constraints implied from its context.

For example, this declaration:

```
array [1..2] of var set of 0..9: x = [{0}, _];
```

becomes:

```
var set of 0 .. 9: u::var_is_introduced;
array [1..2] of var set of 0..9: x = [{0}, u];
```

```
----------------
_::A  -->  T: X::A::var_is_introduced; \ X::A

Annotations rationale:
- A: maybe-copy-if-modified (the expression), maybe-copy-to-new-variables (the
  declaration).
- var_is_introduced: mark-new-variables.
----------------
```

2.2.8 Conversion to FlatZinc built-ins.

This step converts the remaining MiniZinc built-ins and array accesses into FlatZinc built-ins. The FlatZinc built-ins may be type-specialised, and will be reified if the MiniZinc built-in occurs inside a Boolean expression (other than conjunction).

The most complex cases involves linear (in)equalities. Each one is replaced with a (type-specific, possibly reified) linear predicate, unless it can be replaced with a simpler arithmetic constraint.

The next case is that array accesses involving non-fixed indices are replaced with element constraints. Section 3.3 describes this in detail.

After that, conjunctions and disjunctions of identifiers (such as those that might be produced by reification) can be replaced with the N-ary conjunction and disjunction constraints `array_bool_or` and `array_bool_and`. This step is not strictly necessary as it can be emulated with binary conjunction and disjunction, but it may produce more compact FlatZinc models that can be solved faster.

If a predicate that lacks a body needs to be reified, a `_reif` suffix is added to it along with the extra Boolean argument. If no predicate with that name exists, it is a translation-time abort.

The remaining cases are simpler. Section 3 specifies them in detail.

```
----------------
(F1(E1::A1, F2(E2::A2)::B2)::B1 = E3::C)::D  -->  G(E1, E2, E3)::D

Note.  For similar built-ins, this translation step applies in the obvious
manner.

Annotations rationale.
- A1, A2, B1, B2, C:  ad hoc.  Copy-if-copied doesn't apply, because
  FlatZinc doesn't support annotations on arguments to builtins.  XXX: hard
  to say what is right.  Most sensible that either they're all copied to the
  same position as D, or none are copied.
- D: maybe-copy-if-modified.
----------------
```

For example, our running example becomes:

```
(s[0] + (-1)*s[2] <= -2) <-> b1
```

becomes the reified FlatZinc built-in:

```
int_lin_le_reif([1,-1], [s[0], s[2]], -2, b1)
```

Similarly, a linear expression in a solve item can be converted to a call `int_float_lin`.

2.2.9 Top-level conjunction splitting.

This step splits top-level conjunctions into multiple constraint items.

```
----------------
constraint (((E1::A1 /\ E2::A2)::B1 /\ E3::A3)::B2 /\ ... )::B3;
    -->  constraint E1::A1::B1::B2::B3;
         constraint E2::A2::B1::B2::B3;
         constraint E3::A3::B2::B3;
         ...

Annotations rationale.
- A1, A2, ...: maybe-copy-if-copied.
- B1, B2, B3, ...: maybe-copy-multiple.
----------------
```

2.2.10 Annotation declaration removal.

This step removes any annotation declarations.

```
----------------
annotation F(T1: X1, ...);  -->  <empty>

Annotations rationale.
- None needed.
----------------
```

2.3 Variations

These transformations provide a translation that is standard while also supporting much of what a solver writer would want. They are clearly not ideal for every underlying solver. For example, solvers may be more efficient on the undecomposed versions of Boolean constraints or non-linear constraints. It would be good to be able to control which transformations should be applied for a particular solver once we have more experience.

3 MiniZinc Built-ins

This section gives full details on how the MiniZinc built-in operations are translated to FlatZinc.

3.1 Compound Built-ins

Compound built-ins are unrolled into sequences of simpler MiniZinc operations, which are then translated further.

- `sum` is unrolled with `+`.

- `product` is unrolled with `*`.
- `forall` is unrolled with `/\`.
- `exists` is unrolled with `\/`.
- `array_union` is unrolled with `union`.
- `array_intersect` is unrolled with `intersect`.

3.2 Atomic Built-ins

Atomic built-ins with a fixed return value are evaluated and replaced with a value.

The remainder are replaced with FlatZinc built-ins according to the list below. For example, `x * y = z` is replaced with `int_times(x, y, z)`.

The comment after each type-inst signature indicates briefly how it is handled ("`--> x`" indicates that a built-in is renamed to "`x`"). Some of them will be converted to the reified version if they occur inside a Boolean expression other than conjunction.

```
var bool: '<'(var int,        var int)        % --> int_lt
var bool: '<'(var float,      var float)      % --> float_lt
var bool: '<'(var bool,       var bool)       % --> bool_lt
var bool: '<'(var set of int, var set of int) % --> set_lt
% Similarly:
%   '>'      --> *_gt
%   '>='     --> *_le
%   '<='     --> *_ge
%   '=='/'=' --> *_eq
%   '!='     --> *_ne

var int:   '+'(var int,   var int)        % --> int_plus    or int_lin_*
var float: '+'(var float, var float)      % --> float_plus  or float_lin_*
var int:   '-'(var int,   var int)        % --> int_minus   or int_lin_*
var float: '-'(var float, var float)      % --> float_minus or float_lin_*
var int:   '*'(var int,   var int)        % --> int_times   or int_lin_*
var float: '*'(var float, var float)      % --> float_times or float_lin_*

var int:   '+'(var int  )                 % removed
var float: '+'(var float)                 % removed
var int:   '-'(var int  )                 % --> int_negate
var float: '-'(var float)                 % --> float_negate

var int:   'div'(var int,   var int)      % --> int_div
var int:   'mod'(var int,   var int)      % --> int_mod
var float: '/'  (var float, var float)    % --> float_div

var int:   min(var int,   var int)        % --> int_min
var float: min(var float, var float)      % --> float_min
var int:   max(var int,   var int)        % --> int_max
var float: max(var float, var float)      % --> float_max

var int:   abs(var int  )                 % --> int_abs
var float: abs(var float)                 % --> float_abs
```

```
var bool: '/\' (var bool, var bool)       % --> two constraint items or
                                          %     bool_array_and or bool_and
var bool: '\/' (var bool, var bool)       % --> bool_array_or or bool_or
var bool: '->' (var bool, var bool)       % --> bool_right_imp
var bool: '<-' (var bool, var bool)       % --> bool_left_imp
var bool: '<->'(var bool, var bool)       % --> bool_eq
var bool: 'xor'(var bool, var bool)       % --> bool_xor

var bool: 'not'(var bool)                 % --> bool_not

var bool: 'in'(var int,  var set of int)          % --> set_in

var bool: 'subset'  (var set of int, var set of int)  % --> set_subset
var bool: 'superset'(var set of int, var set of int)  % --> set_subset

var set of int: 'union'    (var set of int,
                            var set of int)       % --> set_union
var set of int: 'intersect'(var set of int,
                            var set of int)       % --> set_intersect
var set of int: 'diff'     (var set of int,
                            var set of int)       % --> set_diff
var set of int: 'symdiff'  (var set of int,
                            var set of int)       % --> set_symdiff

var int: card(var set of int)                     % --> set_card

string: show(_)               % --> show (i.e. unchanged)
string: show_cond(_, _, _)    % --> show_cond (i.e. unchanged)
string: '++'(string, string)  % --> a comma in an array literal, e.g.
                              %         ["x = " ++ show(x)]
                              %     --> ["x = ",   show(x)]
```

3.3 Array Accesses

Array accesses with non-fixed indices are converted to element constraints. Here, on the left-hand side, we represent the type-inst signature of array accesses with the name `'[]'`, but this is not valid MiniZinc.

```
'[]'(array[int] of     bool,  var int) -> var bool
                                          % --> array_bool_element
'[]'(array[int] of var bool,  var int) -> var bool
                                          % --> array_var_bool_element
'[]'(array[int] of     int,   var int) -> var int
                                          % --> array_int_element
'[]'(array[int] of var int,   var int) -> var int
                                          % --> array_var_int_element
'[]'(array[int] of     float, var int) -> var float
                                          % --> array_float_element
'[]'(array[int] of var float, var int) -> var float
                                          % --> array_var_float_element
'[]'(array[int] of     set of int, var int) -> var set of int
                                          % --> array_set_element
'[]'(array[int] of var set of int, var int) -> var set of int
                                          % --> array_var_set_element
```

4 Syntax Differences Between the Languages

FlatZinc's syntax is mostly a subset of MiniZinc's, in the sense that any syntactically-valid FlatZinc model is a syntactically-valid MiniZinc model. (But the two languages have different built-ins, so any non-trivial FlatZinc model will not be a valid MiniZinc model.)

Nonetheless, there are some subtleties. The following list covers some potential pitfalls that may trip someone writing a MiniZinc-to-FlatZinc translator.

- MiniZinc items can appear in any order. FlatZinc items can not: variable declarations items must precede constraint items, which must precede the solve item, which must precede (if present) the output item.
- In FlatZinc, call expressions (e.g. `int_lt(a, 3)`) can only appear at the top level of constraint items. They cannot be used as sub-expressions. (Hence the "Flat" in "FlatZinc".)
- In MiniZinc, annotations can be attached to any expression. In FlatZinc, annotations can only be attached to call expressions. E.g. in `int_lt(a, 3)` annotations can be attached to the call expression but not the `a` or `3` expressions.
- In MiniZinc, objective functions in solve items can be arbitrary expressions. In FlatZinc they are more restricted.
- FlatZinc output items are much more restrictive than MiniZinc output items.
- In MiniZinc, in all lists, the comma or semi-colon separator can also be used as a terminator. For example, in a function call `foo(a,b,c)` is equivalent to `foo(a,b,c,)`. This is so that multi-line lists can have a separator after every line, which can make code more consistent and easier to edit. For example:

  ```
  set of int: s = {
      1,
      2,
      3,
  };
  ```

  FlatZinc does not allow this; commas are always separators, and `foo(a,b,c,)` is a syntax error.